ZIL

Marc S. Blank

October 1982

Contents

1 The Z System

The Z System is composed of the various modules which are used to create INTERLOGIC games. At the highest level is Z Implementation Language (ZIL), which is an interpreted language running under MDL. Since ZIL is a MDL subsystem, all of the debugging features of MDL itself can be used in the creation and debugging of INTERLOGIC games. ZIL code is run through the ZIL Compiler (ZILCH) producing Z Assembly Language code which is, in turn, assembled by the Z Assembler Program (ZAP) into machine-independent Z-codes. These Z-codes can be run on any target machine which sports a Z-machine emulator (ZIP).

The author of an INTERLOGIC game need not be familiar with the workings of the compiler, assembler, or emulators to a great extent. The compiler does have a few idiosyncrasies, however, which will be noted as necessary. The remainder of this manual describes MDL and ZIL, starting with simple concepts but eventually describing the full power of the system.

2 Writing in ZIL

MDL (pronounced MUDDLE) is the host language for ZIL and knowledge of MDL is an advantage in learning the ZIL system. However, there are a number of important restrictions and simplifications built into ZIL; it is therefore important that even seasoned MDLers read this section.

2.1 ZIL TYPEs

The ZIL world contains a relatively small number of classes of objects and these classes are called TYPEs. Every operation in ZIL expects to receive objects of specific TYPEs as arguments. For simplicity, 'a FOO' will be used as a shorthand for 'an object of TYPE FOO'.

2.1.1 FORM

FORMs are represented as a collection of other objects (of any type) surrounded by balanced angle-brackets. FORMs are used to perform the various operations in the ZIL world. These operations may be either built-in subroutines or user-defined ROUTINEs. Here are some FORMs:

`<+ 1 2>` This adds the integers 1 and 2.

`<SET A 10>` This sets the local variable A to the integer 10.

`<G=? <+ 4 1> 10>` This returns TRUE if the sum of 4 and 1 is greater-than or equal to 10.

`<SETG FOO <* ,COUNTER <RANDOM 4>>>` This sets the global variable FOO to the product of the value of the global variable COUNTER and a random number from 1 to 4.

The first element of a form indicates the operation to be performed and all other elements are the arguments to that operation.

2.1.2 Prefix Notation

Prefix notation, sometimes referred to as Polish notation, is different from the infix notation of ordinary arithmetic and reverse-Polish notation of some calculators. Below are some examples of equivalent expressions in infix and prefix notation:

```
4 + 7
<+ 4 7>

8 - 6
<- 8 6>

9 + (4 * 6 - 6 / 3)
<+ 9 <- <* 4 6> </ 6 3>>>
```

It may take some time to become accustomed to prefix notation. One thing to keep in mind is the balancing of brackets. Notice that with prefix notation an operator can take an arbitrary number of arguments and that the nesting is never ambiguous (i.e. the parentheses of infix notation are not necessary). In addition, operator precedence can be completely ignored.

2.1.3 Nested Expressions

As can be seen from the previous examples, it is possible to nest expressions. In fact, there is no limit on the depth of the nesting. FORMs are evaluated from left to right, top to bottom.

2.1.4 FIX (Integer)

Objects of TYPE FIX represent integers in the range -32767 to 32767 and are always represented in decimal. Integers of greater magnitude are illegal. Floating point numbers are not allowed in ZIL.

The following operations require two arguments, both FIXes: + (addition), - (subtraction), * (multiplication), / (division), and MOD (modulus). Each returns the appropriate FIX.

In addition, ABS requires one FIX and returns its absolute value, and RANDOM, given a FIX, returns a FIX between one and that FIX, inclusive.

There are three predicates which operate on pairs of FIXes: L? (less than), G? (greater than), and ==? (equal to). In addition, the predicate 0? (equal to zero?) takes a single FIX. All predicates return a 'true' value or a 'false' value. See the section below on conditionals for a full description of truth in the ZIL sense.

Here are some examples of the use of FIXes:

```
<+ 10 20>
<+ </ 10 2> 1>
```

2.1.5 ATOM (Variable)

ATOMs can be thought of as variables. Their names can be almost anything, but safest is a combination of capital letters, numbers, hyphens, question marks, and dollar signs (e.g. FOOBAR, V-WALK, V-$VERIFY). ATOMs can be thought of as coming in two varieties: LOCAL and GLOBAL.

LOCAL ATOMs are used as temporary variables within ROUTINEs (i.e. pieces of code). A LOCAL ATOM can be used in any number of ROUTINEs and there are NO conflicts when one routine with LOCAL ATOM X calls another routine with its own LOCAL ATOM X. Each LOCAL ATOM must be explicitly created within the ROUTINE in which it is used. The mechanism by which this is done is described in a following section. To set the value of a LOCAL ATOM within a ROUTINE, one says:

```
<SET atom-name value>
```

where 'atom-name' and 'value' correspond to an ATOM and an arbitrary value. To retrieve the value of a LOCAL ATOM, one says:

```
.atom-name
```

(period followed by 'atom-name') where 'atom-name' is the ATOM whose value is required.

GLOBAL ATOMs have values which correspond to: rooms and objects, ROUTINEs, flags, properties, variables and tables. The value of a GLOBAL ATOM is accessible to all ROUTINEs at all times. To set the value of a GLOBAL ATOM, one says:

```
<SETG atom-name value>
```

analogously with SET. To retrieve the value of a GLOBAL ATOM, one says:

```
,atom-name
```

(comma followed by 'atom-name').

As will be seen later, OBJECTs, flags, and properties are set up during OBJECT creation and are not explicitly SETG'ed.

2.1.6 STRING

STRINGs are what would be called 'character strings' in other languages. They are used exclusively for printed text. They are represented by a series of characters surrounded by double-quotes. If a double-quote is necessary in the STRING, it must be preceded by a backslash. Here are some strings, followed by their printed representation:

```
"Hello, there!"  -->  Hello, there!

"The man says \"Foobar!\""  -->  The man says "Foobar!"
```

2.1.7 LIST

LISTs are represented as a series of other objects surrounded by matching parentheses. These are used within ROUTINEs for purposes of clarity (seeing angle-brackets everywhere would be downright disorienting). Their use will be described later.

2.1.8 TABLE

A TABLE is what might be referred to as an array in other languages. TABLEs are structures containing arbitrary elements (e.g. OBJECTs, FIXes, STRINGs, etc.). They must be created at 'top-level' (i.e. NOT within a ROUTINE), as follows:

```
<TABLE element element element ...>
```

A special kind of TABLE whose initial element is the number of other elements in the TABLE is created as follows:

```
<LTABLE element element element ...>
```

Note that the first element in this declaration is NOT the number of other elements; that number will be automatically generated.

In ROUTINEs which need to know the length of a TABLE (e.g. a general routine which must search through a TABLE or one which randomly picks an element from within a TABLE), LTABLE must be used. For TABLEs of known size, LTABLE is not necessary.

By convention, the first element of a TABLE is element zero. To retrieve an element from a TABLE, use:

```
<GET table element-number>
```

To place an element within a TABLE, use:

```
<PUT table element-number>
```

2.1.9 OBJECTs

OBJECTs correspond, in the game environment, to objects, rooms, and characters (including the player). The creation of OBJECTs is described below, but the operations used for their manipulation are described here.

Any OBJECT may have, at most, one container and any number of contents. An OBJECT's initial container is determined when it is created. The location of an OBJECT can be returned by:

```
<LOC object>
```

Similarly, one may determine whether or not an OBJECT is in another particular OBJECT by saying:

```
<IN? object-1 object-2>
```

which checks whether or not object-1 is contained within object-2.

OBJECTs can be placed inside other OBJECTs using MOVE:

```
<MOVE object-1 object-2>
```

which moves object-1 into object-2. An OBJECT may be moved into never-never land (i.e. it can be made to have no container, equivalent to being nowhere) with REMOVE:

```
<REMOVE object>
```

To find the contents of a given OBJECT is a bit unusual. The 'first' OBJECT contained in a given OBJECT can be found with:

```
<FIRST? object>
```

Note that FIRST? ends with a question mark, indicating that it is a predicate. If there is nothing contained in 'object', FIRST? returns 'false'.

Other OBJECTs within a given OBJECT can be found using NEXT? as follows:

```
<NEXT? object>
```

NEXT? is defined thus: it returns the 'next' OBJECT which is ALSO contained in the OBJECT's container. Ain't it confusing? Notice that like FIRST?, NEXT? is a predicate. If there is no 'next' OBJECT, it returns 'false'. As an example, let's assume that there is an object X containing objects Y and Z. FIRST? of X will be Y. NEXT? of Y will be Z. NEXT? of Z will be 'false'. (NEXT? of X is unknowable from this example.)

OBJECTs may also have up to 32 condition flags, most of which are designed into the substrate of the game. Among these are OPENBIT (whether a door or container is presently open), TAKEBIT (whether an OBJECT can be taken), DOORBIT (whether an OBJECT is a door), and ONBIT (whether an OBJECT is a source of light). The initial state of these flags is determined during OBJECT creation, and all of the substrate-contained flags are described later.

To check whether a given flag is 'on', use:

```
<FSET? object flag>
```

To set a flag (i.e. turn it on) and clear a flag (i.e. turn it off), use:

```
<FSET object flag>
<FCLEAR object flag>
```

As was noted earlier, OBJECTs and flags are GLOBAL ATOMs; therefore, the following might appear in game code:

```
<FSET ,AIRLOCK-DOOR ,OPENBIT>
```

This would cause the AIRLOCK-DOOR to be considered 'open'. The ramifications of this particular example would include the ability to go through the door without obstruction and the ability to look through the door. The ramifications of the various flags is discussed below.

OBJECTs also have up to 31 'properties', whose values are explicitly manipulated relatively infrequently during game writing. Among these properties are SIZE (weight of an OBJECT), CAPACITY (total weight that a container can hold), ACTION (ROUTINE to be called for special case actions), and LDESC (long description). These properties are set up during OBJECT definition and are described completely later on. It should be noted, however, that new properties cannot be added to an OBJECT while a game is running. If a property is needed for a particular OBJECT, it must be initialized when the OBJECT is defined.

Retrieving the value of a property for a given OBJECT is done with:

```
<GETP object property>
```

Similarly, setting the value of a property for a given OBJECT is done with:

```
<PUTP object property value>
```

Like condition flags, properties are GLOBAL ATOMs. In the context of GETP and PUTP, their names must be prefixed by the letter P and a question-mark. In other words:

```
<GETP ,RUSTY-KNIFE ,P?SIZE>
```

would retrieve the SIZE property of the RUSTY-KNIFE.

If, in a call to GETP, the supplied OBJECT does not have the supplied property defined, the result is 'false'. This default result can be altered, if it is so desired, by placing a statement like this in the ZIL 'load file' (see further on):

```
<PROPDEF property-name default-value>
```

where 'property-name' is the name of the property for which a default will exist, and 'default-value' is the value which GETP will return if the property is not defined for a given OBJECT. Here's an example:

```
<PROPDEF SIZE 5>
```

2.2 Conditional Expressions

One cannot discuss conditional expressions without explaining the meaning of 'truth'. ZIL has a rather simplistic view of truth: anything which is not zero is true. For historical reasons, a distinction is made between 'false' and zero and it is the cause of some confusion. GLOBAL ATOMs are frequently used to save the state of a global condition (e.g. SUIT-ON? might be 'true' or 'false' depending on whether one is wearing the spacesuit). Two special tokens are used to mean 'true' and 'false' in these cases: T and <> (open followed by closed angle bracket). In ZIL code, therefore, one should use:

```
<SETG SUIT-ON? T>
<SETG SUIT-ON? <>>
```

rather than:

```
<SETG SUIT-ON? 1>
<SETG SUIT-ON? 0>
```

This distinction makes code more understandable to the casual observer: in the former example, it is clear that SUIT-ON? is a condition flag. In the latter, it is unclear whether SUIT-ON? is a condition flag or a variable whose value is a FIX.

Some of the most common operations in ZIL are predicates and return one of two values: true (not zero, usually one) and false (zero). We have already described some of these: L?, G?, and ==? (for arithmetic) and IN? and FSET? (for OBJECTs).

The other operations dealing with conditionals are mentioned here.

2.2.1 EQUAL?

EQUAL? takes from two to four arguments and determines whether the first is equal to any of the other arguments.

```
<EQUAL? .OBJ ,LANTERN ,CANDLES>
```

The example checks to see whether the value of the LOCAL ATOM OBJ is either the LANTERN or the CANDLES (presumably these are OBJECTs). EQUAL? returns T or <> (i.e. 'true' or 'false'). It can be used with any TYPEs of arguments. Thus,

```
<EQUAL? .NUM 1 2 3>
```

could be used to check whether or not the LOCAL ATOM NUM was equal to one, two, or three.

2.2.2 NOT

NOT takes one argument. If it is not 'false', it returns 'false'. If it IS 'false', it returns 'true'. Thus, if the LOCAL ATOM OBJ is 12, then

```
<NOT .OBJ>
```

will return 'false'. To restate in another way, NOT returns 'true' only if its argument was 'false'. It returns 'false' in every other case.

2.2.3 AND

AND takes any number of expressions and evaluates them from left to right. It returns 'true' only if ALL of the expressions are 'true'. Otherwise it returns 'false'. For example,

```
<AND <G? .NUM 10> <L? .NUM 20>>
```

returns 'true' if the value of LOCAL ATOM NUM is BOTH greater than 10 and less than 20. Otherwise, it returns 'false'.

2.2.4 OR

OR, similar to AND, takes any number of expressions and evaluates them from left to right. However, it returns 'true' if ANY of the expressions is 'true'. Otherwise, it returns 'false'. For example,

```
<OR <L? .NUM 11> <G? .NUM 19>>
```

returns 'true' if the value of the LOCAL ATOM NUM is EITHER less than 11 OR greater than 19. Otherwise, it returns 'false'. (This is the opposite of the example for AND.)

2.2.5 COND

By now, those unfamiliar with MDL have had about enough of conditionals returning 'true' and 'false' and are probably wondering just who it is who decides to do something depending on those values. The answer is COND, probably the most commonly used operation in the language. The structure of a COND expression is:

```
<COND (predicate expression expression ...)
      (predicate expression expression ...)
      ...
      (predicate expression expression ...)>
```

First note that COND, like AND and OR, takes any number of arguments, which are all LISTs. Each LIST contains at least two elements: a predicate (i.e. conditional expression) and something(s) to do if that predicate returns 'true'.

COND works like this: starting with the first LIST, it evaluates the predicate. If it returns 'true', then ALL of the remaining expressions are evaluated in turn, and the COND itself returns the value of the last of those expressions. If it returns 'false', then the next LIST is examined in the same way. If none of the predicates returns 'true', then the COND itself returns 'false'. Here is a typical example of a COND expression:

```
<COND (<VERB? TAKE>
       <TELL "Your hand passes through its object." CR>)
      (<VERB? SCORE>
       <TELL "How can you think of your score now?" CR>)>
```

This COND may as well have been lifted directly from one of the ZORK games. VERB?, one would assume, is a predicate of some sort, returning 'true' or 'false' depending on its argument(s). In fact, it is conventional to name ROUTINEs which return only 'true' or 'false' with a trailing question mark. It happens that VERB? compares its argument(s) with the parser's idea of which action was requested by the player (see far later on for a description of the parser's ideas about everything). If they match, it returns 'true'. Otherwise, you guessed it. The TELL operation is used for constructing printed text. Not surprisingly, the 'CR' means that a carriage return is printed after the STRING (much more on TELL later).

It has been said that ZORK is one giant COND. This may be exaggeration, but it hits near the mark. Here are some more partial examples:

```
<COND (<NOT ,SUIT-ON?>
       <JIGS-UP "You can't breathe vacuum and thus die.">)>
```

Here is a case in which NOT is frequently used. In this example, the ROUTINE JIGS-UP is called if the condition flag SUIT-ON? is NOT 'true' (i.e. the suit is not being worn). JIGS-UP means more or less that.

Predicates can be (and often are) built up using ANDs and ORs. Thus,

```
<COND (<AND <NOT <EQUAL? .OBJ ,WEASEL ,RAT-ANT>>
    ,SUIT-ON?>
       <TELL "He spits on your suit." CR>)>
```

The SET and SETG operations can often be nested directly into conditional expressions (sometimes the compiler will do the wrong thing), like this:

```
<COND (<SET OBJ <FIRST? .OBJ>>
       <TELL "Something or other." CR>)>
```

3 ROUTINEs

ROUTINEs are user-constructed subroutines which are the backbone of the INTERLOGIC games. The main loop, parser, verb handlers, special case handlers for OBJECTs, and the like are all ROUTINEs. Each ROUTINE has a 'name', which is a GLOBAL ATOM.

A ROUTINE is defined as follows:

```
<ROUTINE name (argument-list ...) expression expression ...>
```

where 'name' is a legal ATOM name, 'argument list' will be described later on, and 'expression' is any legal ZIL expression. When a ROUTINE is called, each of the expressions is evaluated in turn and the result of the last evaluation is the value of the call to the ROUTINE.

3.1 Argument List

The argument list is a LIST which can be thought of as coming in three parts, none of which are required. The first part are LOCAL ATOMs corresponding to required arguments to the ROUTINE. Some examples:

```
<ROUTINE MOVE-ALL (LOC1 LOC2) ....>

<ROUTINE MOUSE-F () ...>
```

In the first example, a ROUTINE named MOVE-ALL is defined to take exactly two arguments. Within the context of MOVE-ALL, the LOCAL ATOMs LOC1 and LOC2 will have the value of the arguments passed to MOVE-ALL. The ROUTINE MOUSE-F takes no arguments. Here is a typical call to MOVE-ALL:

```
<MOVE-ALL .CONT ,TRASH-BIN>
```

In this case, LOC1 (in MOVE-ALL) will have the value of the LOCAL ATOM CONT in the calling ROUTINE and LOC2 (again, in MOVE-ALL) will have the TRASH-BIN as a value. Note that calling a ROUTINE is no different than calling a built-in subroutine such as COND, +, or FCLEAR.

In addition to required arguments, ROUTINEs may have optional arguments as well. Indicate that all further arguments are optional by placing the STRING "OPTIONAL" after the ATOMs (if any) representing the required ones. After the "OPTIONAL", each optional argument is indicated by a LIST of two elements: a LOCAL ATOM and a default value (if the argument is not passed by the calling ROUTINE). For example,

```
<ROUTINE ALREADY (STR "OPTIONAL" (OBJ <>)) ...>
```

ALREADY takes one or two arguments, only one of which (STR) is required. The second (OBJ) is not required and, if not passed, will be SET to 'false'.

```
<ALREADY "open">

<ALREADY "open" ,AIRLOCK-DOOR>
```

In the first example above, STR's value will be "open" and OBJ's will be 'false'. In the second, OBJ's value will be the AIRLOCK-DOOR. This ROUTINE is used in STARCROSS, and the result of these examples would be:

```
It is already open.

The airlock door is already open.
```

Oh, why not. Here's the ROUTINE for ALREADY in its full glory:

```
<ROUTINE ALREADY (STR "OPTIONAL" (OBJ <>))
<COND (.OBJ <TELL "The " D .OBJ>)
      (T <TELL "It">)>
<TELL " is already " .STR "." CR>>
```

At the risk of getting too far ahead of the game, it should be clear that TELL is something which prints text; in fact, it cobbles together arbitrary numbers of things into sentences. It takes, as arguments, STRINGs (either 'in person' or as the value of an ATOM), the ATOM CR (which means print a carriage return/line feed), and a few special items. One of these is the ATOM D followed by something which had better be an OBJECT (represented as the value of a LOCAL or GLOBAL ATOM). It means to print the 'short description' of the OBJECT. In the example above, "airlock door" would have been defined as the short description of the AIRLOCK-DOOR OBJECT. If you understand this example, you're doing just fine.

The third part of the argument list (also not required) refer to other LOCAL ATOMs which you wish to use as temporary variables within the ROUTINE. Any LOCAL ATOM you use in a ROUTINE MUST be indicated somewhere within the argument list. To indicate the place in the argument list at which temporary variables start, place the STRING "AUX" at that point. This is followed by any number of either ATOM names or LISTs (exactly like those for optional arguments) containing an ATOM name and a default (i.e. initial) value. It is illegal to retrieve the value of a LOCAL ATOM unless it has either a default value in the argument list or has been explicitly SET within the ROUTINE.

```
<ROUTINE COUNT (OBJ "AUX" (CNT 0)) ...>
```

This ROUTINE which, incidentally, takes exactly one argument, initializes the LOCAL ATOM CNT to zero. Presumably CNT is used to count something, and somewhere in the innards of the ROUTINE are statements like:

```
<SET CNT <+ .CNT 1>>
```

which would increment the value of CNT.

3.2 Names for ATOMs

It is useful, both for yourself and others with the temerity to look at your code, for you to choose your ROUTINE, variable, etc. names to be somewhat mnemonic. If one has a dirty rag in a game, it would be best to call it DIRTY-RAG. A ROUTINE which counts grues might be called COUNT-GRUES. It is common for ROUTINEs to have ATOMS with names like OBJ, CNT, and DIR to refer to OBJECTs, counters, and directions. The easier for you (and others) to remember what things are called, the easier will be the writing and debugging. I remember some anguish over not being able to find the brown rod in the STARCROSS listing because it had (for historical reasons) been named CHIEF-KEY, unlike the others which were called GREEN-KEY, RED-KEY, and other colorful names.

3.3 Looping

Now and again, one wants one's ROUTINE to go into some kind of loop. Searching through TABLEs comes to mind as a prime example. This is done as follows:

```
<REPEAT () expression expression ...>
```

Note that the empty-looking LIST is required, even though it seems to have no purpose (the MDLers reading this are, no doubt, smirking). The expressions are arbitrary and when the last of them is evaluated the whole thing begins over again, time without end. Unless, somewhere within, is a RETURN statement. RETURN (no arguments) exits a loop. Here's a useless loop:

```
<REPEAT ()
<COND (<FSET? .OBJ ,INVISIBLE>
       <SET CNT <+ .CNT 1>>)>
<COND (<NOT <SET OBJ <NEXT? .OBJ>>>
       <RETURN>)>>
```

Without describing the meaning of NEXT? and hoping that in the argument list of whatever ROUTINE this is embedded within there is an initialization of the ATOMs CNT and OBJ, this section of code might count the number of invisible objects somewhere.

3.4 Exiting a ROUTINE

ROUTINEs evaluate all of the expressions after the argument list in order and return the result of the last of these. One can, however, at any point within a ROUTINE, cause the ROUTINE to return a specific value using one of the following:

```
<RTRUE>

<RFALSE>

<RETURN anything>
```

Each of these causes the ROUTINE to immediately stop its execution are return 'true', 'false', and 'anything', respectively. RTRUE and RFALSE turn out to be enormously important. A warning: RETURN used within a REPEAT only causes the REPEAT to terminate. RTRUE and RFALSE, used ANYWHERE, cause the entire ROUTINE to terminate.

3.5 Restrictions in ROUTINEs

The most important of these is that a ROUTINE may take, at most, three arguments (any mix of required and optional). In addition, there can only be sixteen LOCAL ATOMs in a ROUTINE. These restrictions are mentioned for completeness: there are very few cases in which the first is important, and none so far in which the second was important.

3.6 Formatting ROUTINEs

It should be mentioned explicitly that the formatting characters (space, tab, carriage return, line feed, form feed, etc.) are completely ignored by ZIL (and MDL) and can be sprinkled about liberally to make the code appear more readable. It is common practice to 'line up' pieces of code vertically within the more complicated structures (e.g. COND, REPEAT, etc.) so that it is obvious by inspection which pieces of code correspond to which structure. Examination of existing game code should be enlightening in this regard. The importance of good formatting can hardly be overstated. Code which is not formatted or formatted poorly is completely unreadable.

4 Printing and TELL

As was noted earlier, TELL is used to cobble together printed text. TELL takes any number of arguments which are one of the following:

- Explicit STRINGs
- STRINGs which are the value of an ATOM
- STRINGs which are the value of an arbitrary expression, including a call to another ROUTINE
- The ATOM D followed by an OBJECT (represented as the value of an ATOM)
- The ATOM N followed by a FIX (either explicitly or as the value of an ATOM)

Here is a complicated example:

```
<TELL "You are in " D .LOC ". You have " N ,SCORE " points." CR>
```

This might print:

```
You are in Red Hall. You have 37 points.
```

supposing that LOCAL ATOM LOC's value was the RED-HALL OBJECT and that GLOBAL ATOM SCORE's value was the FIX 37.

5 PARSER 101

This is intended to be an introduction to the world of the PARSER, the most incomprehensible piece of code in the INTERLOGIC system. A subsequent section will concern itself with advanced parser concepts.

It is best to think of the parser as a black box with from one to three outputs: an action to be performed, a direct object, and an indirect object. Each of the three corresponds to a GLOBAL ATOM: PRSA, PRSO, and PRSI, respectively. Very frequently, the ACTION ROUTINE for an OBJECT needs to check on which action is being performed. For this case, a special operation has been constructed, namely the predicate VERB?. Here's an example:

```
<VERB? WALK SCORE TAKE PUSH OPEN>
```

This expression will return 'true' if the action requested was one of WALK, SCORE, TAKE, PUSH, or OPEN. Otherwise, it will return 'false'.

6 Main Loop

Understanding the main loop is absolutely vital in designing games. The flexibility and complexity allowed in INTERLOGIC games are almost wholly due to the parser/main loop combination.

6.1 Parser and Philosophy

The main loop starts with a call to the parser. If the parser 'fails' (e.g. a word was unknown, the sentence didn't make sense, etc.), nothing further happens and the loop is restarted. If it succeeds, the three ATOMs described in the preceding section are SETG'ed and a number of ROUTINEs are called in turn to determine the result of the player's intended action.

Before indicating the order and purpose of the 'number of ROUTINEs', it is important to understand the idea of 'handling'. In ZIL, we say that a ROUTINE 'handled' the player's action if it finishes all of the processing required for that action. Each ROUTINE decides whether or not it considers the action 'handled', and its decision is final. A ROUTINE 'handles' an action simply by returning a 'non-false' (i.e. anything other than a 'false').

It is a given in the INTERLOGIC world that no successfully parsed user input should leave the user without an indication of the result of that action. The following should never happen:

```
>kill the troll

>
```

Someone, somewhere, should have 'handled' the requested action. This 'someone, somewhere' is defined to be the default action handler, the last step in the chain of called ROUTINEs. It MUST print something, and that thing should be the default response for that type of action. For example, the default action handler for EAT might do this:

```
<TELL "I don't think the " D .OBJ " would agree with you." CR>
```

In fact, it does. If EAT for a specific OBJECT wants to do something different, then the OBJECT itself must supply a ROUTINE to 'handle' the EAT action. An OBJECT supplies such a ROUTINE by having an ACTION property (see the section on OBJECT properties, above).

6.2 Basic Handler Sequence

These are the basic steps in the 'handler' sequence:

Preaction: This ROUTINE is intended to short circuit the remainder of the sequence if some prerequisite for the intended action is not met. There may be a preaction ROUTINE associated with any syntax (see later on for definition of syntaxes). For example, the ROUTINE called PRE-PUT checks to see whether the object to be put into some other object is something which is takeable. This would prevent any number of container ACTION ROUTINEs from doing precisely the same thing.

Indirect Object: A ROUTINE specified as the ACTION property of an OBJECT.

Container: A ROUTINE specified as the CONTFCN property of the PRSO's container. This is used only rarely.

Direct Object: A ROUTINE specified as the ACTION property of an OBJECT.

Default Action: The default ROUTINE for a given action. An example of this was given for the EAT action previously.

Some important things need to be noted about these previous steps:

- The order of the steps is crucial. The fact that the indirect object ROUTINE is called BEFORE the direct object ROUTINE has implications which have caused some confusion in the past. For example, if the input sentence was "PUT THE EGG IN THE NEST", and there were an ACTION for the nest which handled putting things into it and an ACTION for the egg which handled putting it into things, the nest's ACTION would take precedence. This might not, however, be the thing intended.
- There are cases in which an ACTION ROUTINE must be careful to check whether the OBJECT it 'represents' is the direct object or the indirect object. For example, the parser decides that the sentence "TAKE SWORD FROM STONE" implies the action TAKE. So, however, does "TAKE STONE". If the STONE ACTION ROUTINE checks only for the action being TAKE (e.g. <VERB? TAKE>), it will generate the same response for the two sentences above. This is clearly wrong, and the ACTION ROUTINE should additionally check the value of PRSO as well (e.g. <AND <VERB? TAKE> <EQUAL? ,PRSO ,STONE>>).
- The CONTFCN ROUTINE allows the container of an OBJECT to handle actions on the OBJECT itself. This ROUTINE takes no arguments. Its use is usually to handle special cases involving removing the OBJECT from its container. For example, the nest in STARCROSS contains odds and ends which cannot be removed without provoking the rat-ants. This could only be implemented using the CONTFCN procedure.

6.3 Advanced Handler Sequence

The four basic steps in the sequence represent the large majority of the cases in INTERLOGIC games. However, there are cases in which these steps do not provide enough flexibility. For example, it is difficult to see how one can short-circuit the WALK action in a specific room without mucking around directly with the default WALK action ROUTINE, which would be considered inelegant (to say the least). More difficult would be to implement the death sequence in ZORK I in which almost every action is handled differently than during 'life'. To have been forced to include special code in each action handler for the case of 'death' would have been incredibly tedious.

The solution is to place another sequence around the basic one. This makes the main loop look like the following:

- ACTOR ROUTINE. A ROUTINE which may be associated with each OBJECT in the game which can perform actions. In ZORK I, this would be the game player only. In STARCROSS, however, there are the game player, computer, and various aliens who can perform actions.
- Location ROUTINE. A ROUTINE associated with the location in which the ACTOR is acting. Note that 'location' need not be a ROOM, since the ACTOR may be in a vehicle. See later for this unusual case.
- Basic Handlers. As mentioned above.

Note first that the Location and ACTOR ROUTINEs are ACTION properties of the corresponding OBJECT (in INTERLOGIC games, all of these things are OBJECTs) and are set up when the OBJECT is created.

The ACTOR ROUTINE might be used, for example, to implement the death sequence in ZORK I. The Location ROUTINE might be used to implement an area (say, floating in space) in which WALKing is not possible. The ACTOR and Location ROUTINEs 'handle' an action in precisely the same way as do the basic handlers, i.e. by returning 'true'.

When the entire sequence is finished, the Location ROUTINE is called again and its result is ignored. It might, for example, decide that anything left on the ground at the end of an action is swept away.

6.4 CLOCK

When all of the above steps are finished, the CLOCK runs. Each move may be thought of as one tick of the CLOCK. It is possible to cause arbitrary events to happen at arbitrary times during a game (the mechanism is described later) and all queued events which have come due are processed at this step. After this step, the main loop repeats.

7 ACTION ROUTINEs

There are some conventions to writing ACTION ROUTINEs which will be mentioned here. Some may be obvious from the preceding discussion, others not.

A ROUTINE for an OBJECT (except as noted below) or ACTOR takes no arguments. This is a consequence of the fact that each may be called only once during the main loop and the purpose of the call is, therefore, unambiguous.

The ROUTINE for a Location (ROOM or vehicle) however, takes exactly one argument. By convention, it becomes the value of the LOCAL ATOM RARG. This argument is used to determine the context of the call to the ROUTINE. Two such contexts have been described: the one following the call to the ACTOR ROUTINE, and the one following the result of the entire action. A few others will be described later. Each context has a code (a FIX, really) associated with it. Rather than memorize the FIX associated with each context, a GLOBAL ATOM for each exists whose value is the context code. The codes for the two contexts mentioned here are the values of GLOBAL ATOMs M-BEG and M-END, respectively.

```
<ROUTINE ROOM-F (RARG)
 <COND (<EQUAL? .RARG ,M-BEG> ...)
       (<EQUAL? .RARG ,M-END> ...)>>
```

The preceding might be the shell of the ACTION ROUTINE for a certain ROOM. REMEMBER: The result of the call to a ROOM's ACTION ROUTINE in the M-BEG case is important! If 'true', no further processing occurs.

There is no requirement for names of ACTION ROUTINEs, but it is best to be consistent. My personal suggestion is that the ROUTINE for an OBJECT called FOO should be FOO-F, but to each his own. Having a consistent method insures that it will be easy to locate a piece of code within a large file of ROUTINEs.

8 The Parser and Objects

This discussion deals with 'objects' in the within-the-game sense (i.e. not about 'rooms' and the like) and the way in which the parser understands references to them. An understanding of this topic is necessary in writing OBJECT and ROOM definitions, which is the subject of the following sections.

Every object which can be referenced within a game may have from one to four SYNONYMs and from zero to eight ADJECTIVEs. Whenever the parser recognizes a noun phrase, it looks at all of the currently accessible OBJECTs (the definition of 'accessible' is described in the next section) for one which matches the nouns and adjectives within that phrase. If more than one adjective is supplied by the game player, the first is used (there are exceptions which will be described later). Two nouns can be used within a noun phrase ONLY if separated by the word "OF" (i.e. "PAIR OF CANDLES"). In this case, the second noun is used and the first is ignored. Typically, one would have both PAIR and CANDLES be SYNONYMs of the 'pair of candles' OBJECT; then, one could refer to them either by "CANDLES", "PAIR", or "PAIR OF CANDLES".

A word can be used as both a noun and an adjective in INTERLOGIC games. For example, in DEADLINE, there is a 'bottle of Loblo tablets'. The SYNONYMs for this OBJECT are BOTTLE, LOBLO, and TABLETS. The ADJECTIVE is LOBLO. One might refer to the OBJECT legally as "BOTTLE", "LOBLO", "LOBLO TABLETS", "BOTTLE OF LOBLO TABLETS", "BOTTLE OF LOBLO", "TABLETS", or "BOTTLE OF TABLETS". The flexibility here is quite useful for one might imagine a player using any of these. Naturally, simply saying "BOTTLE" might be ambiguous if there were more than one BOTTLE accessible. If so, the parser would ask the player to choose among the alternatives.

9 Accessibility

In the last section, it was stated that the parser determines which OBJECTs are accessible at any given time. OBJECTs can be divided into three classes, determining the ROOMs in which they can be accessed. LOCAL OBJECTs can be in one place only; these represent ALL objects which can be taken and some others. GLOBAL OBJECTs can be referenced in ALL ROOMs; these represent concepts, names of characters, and the like. LOCAL GLOBAL OBJECTs are objects which can appear in any number of EXPLICITLY specified ROOMs (these ROOMs must specify the LOCAL GLOBALs located therein - see the next section); they usually represent things like doors and geography (rivers, cliffs, trees).

The parser's algorithm for finding objects based on the user's input is to first look at LOCAL OBJECTs in the ROOM or the ACTOR. Only if this fails will it look at LOCAL GLOBALs and GLOBALs. An real example from STARCROSS demonstrates an interesting side effect: There are two buttons, one a LOCAL OBJECT and one a LOCAL GLOBAL OBJECT accessible in the same room. If the user says "PUSH BUTTON", the parser will come back with the LOCAL OBJECT only! Since a LOCAL OBJECT was found, it didn't bother looking elsewhere. Similarly, "PUSH ALL BUTTONS" will still only find the LOCAL OBJECT! The obvious question comes up: why not always check for LOCAL GLOBALs and GLOBALs? The answer is efficiency; it is quite a bit slower to search through the LOCAL GLOBALs and GLOBALs than to look at LOCAL OBJECTs.

Note that "PUSH RED BUTTON" (assuming the red button were the LOCAL GLOBAL) would work, since there is no LOCAL OBJECT matching the adjective and noun. The LOCAL GLOBALs and GLOBALs would then be examined, and the match be made.

10 Creating a ROOM

ROOMs are created through a call to the subroutine ROOM, as follows:

```
<ROOM room-name
(IN ROOMS)
(DESC "short description")
(FLAGS flag-11 ... flag-n)
(property value)
(property value)
...
(property value)>
```

The first four lines are required; 'room-name' and 'short description' are, obviously, variable. The 'room-name' will be the GLOBAL ATOM whose value will be the ROOM. The 'short description' is used in brief room descriptions and on the status line. The 'flags' include RLANDBIT (meaning that the ROOM is on land) and ONBIT (meaning that the ROOM is lit). For purposes of discussion, RLANDBIT should be in every ROOM.

The remaining parts of the room definition are variable and describe such things as the exits from the room, a long description, the ACTION property, and the like. Each will now be described.

10.1 Exits (of all kinds)

Exits come in many flavors, ranging from Vanilla (an unconditional exit, e.g. an always open door) to Bavarian Chocolate Almond Fudge Ripple (an exit which is determined by calling fifteen dozen ROUTINEs). This is how they are described:

10.1.1 Unconditional Exits

Unconditional exits are described thus:

```
(direction TO room-name)
```

where 'direction' is NORTH, SOUTH, EAST, etc. and 'room-name' is the name of the ROOM connected in 'direction'.

10.1.2 Unconditional Non-Exits

An Unconditional Non-Exit (commonly called an 'N-exit') indicates that a certain STRING should be printed if the ACTOR attempts to go in that direction. The only difference between an N-exit and simply not specifying an exit for that direction is what's printed (in the latter case "You can't go that way.").

```
(direction "reason-why-not")
```

where 'direction' is the usual and 'reason-why-not' is a STRING which will be printed.

10.1.3 Conditional Exits

A Conditional Exit (called 'C-exit') allows movement to a given ROOM only if a specified GLOBAL ATOM's value is 'true'. There are two forms of C-exit: one in which the default "You can't go that way." is printed if the condition is 'false' and one in which a specified STRING is printed.

```
(direction TO room-name IF global-atom-name)

(direction TO room-name IF global-atom-name
ELSE "reason-why-not")
```

where 'direction', 'room-name', and 'reason-why-not' are as usual and 'global-atom-name' is the name of the GLOBAL ATOM whose value will be used as the test condition.

10.1.4 Door Exits

A Door Exit (called 'D-exit') allows movement to a given ROOM if the OPENBIT (a condition flag) of the specified door (an OBJECT) is set.

```
(direction TO room-name IF door-name IS OPEN)
```

In this case, 'door-name' is the GLOBAL ATOM used as the name of an OBJECT which had better be a door. If the OPENBIT is not set, the message "The shmoo is closed." is printed, with 'shmoo' replaced by the short description of the door OBJECT.

10.1.5 Flexible Exits

A Flexible Exit (called 'F-exit') is just that. A ROUTINE is called and may do anything it likes. If it returns 'false', it is presumed to have printed a message indicating why the movement failed. Otherwise, it returns the ACTOR's new location.

```
(direction PER routine-name)
```

with 'routine-name' being the name of the ROUTINE to call.

10.1.6 Examples of the various types of exits:

```
(NORTH TO ROUND-ROOM)

(SOUTH "You would be killed by the pounding surf!")

(EAST TO STRANGE-PASSAGE IF CYCLOPS-FLAG)

(EAST TO RED-DOCK IF AIRLOCK-DOOR IS OPEN)

(WEST PER BRIDGE-EXIT-F)
```

10.1.7 Things to remember

The names of directions, the tokens (TO, PER, IF, IS, OPEN, ELSE), and names of ROOMs and ROUTINEs must be capitalized! Remember that ZIL and MDL distinguish case. In fact, the only thing in an exit description which should not be capitalized are the STRINGs.

The substrate files (see later) assume that the standard directions are NORTH, SOUTH, EAST, WEST, NE, NW, SE, SW, UP, DOWN, IN, and OUT. If these directions need to be changed, the appropriate file needs to be updated (this change was needed in STARCROSS to allow PORT, STARBOARD, etc.) Consult Marc or Dave for this.

10.2 ACTION

The ACTION property is specified as follows:

```
(ACTION routine-name)
```

Remember that the ROUTINE for a ROOM takes one argument, whose value is important. Consult the appropriate earlier section to remind yourself.

10.3 GLOBAL

The GLOBAL property is used to define those LOCAL GLOBALs which are accessible in the ROOM. There are a maximum of eight of these.

```
(GLOBAL object-name object-name ...)
```

10.4 LDESC

The LDESC property is the long description of the ROOM. It appears like this:

```
(LDESC string)
```

While we're in the neigborhood, it should be pointed out that the LDESC property is used only (and obviously) for ROOMs whose description does not change during the course of the game. There is another method of long description using the ACTION ROUTINE for the ROOM. In this other case, the LDESC property MUST be excluded from the ROOM definition. There MUST be an ACTION ROUTINE which expects an argument with the value of the GLOBAL ATOM M-LOOK (Aha! There's another one of the context codes mentioned earlier!) For example:

```
<ROUTINE ROOM-F (RARG)
 <COND (<EQUAL? .RARG ,M-LOOK>
<TELL ... CR>)>>
```

11 Room Description (trivia)

We have seen that there are two ways of handling the long description of a ROOM, with LDESC and with an ACTION property. What we haven't seen is just what determines if and when the long description is seen.

INTERLOGIC games come in three levels of verbosity: verbose (meaning that the long description is printed every time a room is entered), brief (meaning that the long description is printed only on the first entry into a room), and superbrief (the long description is never printed). There are commands in the game environment ("VERBOSE", "BRIEF", and "SUPERBRIEF") which control these levels of verbosity.

How does the game know, you are no doubt asking, whether a room has been visited before? The answer is a condition flag called TOUCHBIT, which is used by both objects and rooms. When a room is entered, depending on the game-controlled verbosity level and the state of the TOUCHBIT, a room description is printed; then, the TOUCHBIT is set. There are cases in which a room description changes during the course of the game and the writer wants to call attention to that change. The clever game writer, having read this awe-inspiring document with an almost fanatical dedication, will, at the time the description changes, clear the TOUCHBIT of the ROOM! Incredible nitwits like Meretzky, who can barely read English at all, will fuck up at every point anyways, so why bother explaining?

In case you're still with us, there is an even more arcane piece of room description trivia to mention. M-FLASH is a context code that can be used in a ROOM's ACTION ROUTINE to describe something in the room regardless of the level of verbosity or what's already been printed. It was used twice in ZORK II and has lived in almost total obscurity since.

12 Creating an OBJECT

OBJECTs are created through a call to the subroutine OBJECT (similarly to ROOM), as follows:

```
<OBJECT object-name
(DESC "short description")
(ADJECTIVE adjective-1 adjective-2 ...)
(SYNONYM noun-1 noun-2 ...)
(property value)
(property value)
...
(property value)>
```

The first three lines are required: an OBJECT must have a name, short description, and at least one noun describing it. Most objects have at least one adjective. As was mentioned earlier, there are at most four nouns and eight adjectives describing an OBJECT.

The remainder of the OBJECT definition is variable and describe condition flags, ACTION property, state of containment, and the like. Here they are:

12.1 FLAGS

An OBJECT may have any number of condition flags determined at creation time. All of these can be manipulated during the course of a game. Many of the flags are used within the substrate of the game and need never be manipulated; others are more commonly used.

12.1.1 INVISIBLE

Indicates that the OBJECT is invisible. Invisible objects cannot be referenced; it is exactly as though the OBJECT were not present in its location. Setting and clearing the INVISIBLE flag is pretty much identical to REMOVE'ing and MOVE'ing the OBJECT.

12.1.2 DOORBIT

Says that the OBJECT is a door. This is useful in conjunction with D-exits (described earlier).

12.1.3 CONTBIT

Says that the OBJECT is a container of some kind. The actions for PUT, TAKE, OPEN, and CLOSE (among others) are interested in this.

12.1.4 SURFACEBIT

Means that the container is really a surface. Internally, the games can't tell the difference between a surface and a container. Only this flag distinguishes them.

12.1.5 SEARCHBIT

Means that one can see things inside of things inside the container that has this flag set. The default is that one can only see things through one level of containment. OBJECTs with the SURFACEBIT act as if they have the SEARCHBIT as well.

12.1.6 OPENBIT

Says that the OBJECT is open. This can refer to a door or a container.

12.1.7 TRANSBIT

Indicates that the OBJECT (a container) is transparent. The fellow who prints the description of OBJECTs within ROOMs won't tell the player what's in a container unless it's open or transparent.

12.1.8 NDESCBIT

Says that the OBJECT need never be explicitly described. This is almost always because the OBJECT is described as part of the room description.

12.1.9 READBIT

The OBJECT can be read. This is used in conjunction with the TEXT property and the READ action. (see later, this section).

12.1.10 ONBIT

The OBJECT is a source of light. For ROOMs, this has an analogous meaning (i.e. that the room is self-lit). The room describer and parser determine whether or not a room has light by checking the ONBIT of the ROOM and the ONBITs of the various OBJECTs within the ROOM.

12.1.11 FLAMEBIT

The OBJECT is on fire. The BURN action is interested in this. Note that FLAMEBIT does not (but maybe it should) imply the ONBIT. In general, a torch should be initialized to have both. If it goes out, the ROUTINE which does the dirty work should clear BOTH.

12.1.12 BURNBIT

The OBJECT can be burnt. Mostly these are paper things. The BURN action checks this.

12.1.13 TAKEBIT

The OBJECT can be taken (at least potentially). A number of actions check whether or not an OBJECT is takeable.

12.1.14 TRYTAKEBIT

The OBJECT can't be taken, but its ACTION ROUTINE has a specific response when the player tries to take it. This will cause TAKE's preaction to RFALSE rather than print a default.

12.1.15 VEHBIT

The OBJECT is a vehicle. This is described in too great detail later.

12.1.16 FURNITURE

The object is a piece of furniture on which one can sit. Ask about this.

12.2 LDESC

Similar to the LDESC property in ROOMs, this is the long description of the OBJECT. No LDESC is required and the default long description is "There is a shmoo here." for appropriate 'shmoo'.

```
(LDESC "Long description")
```

12.3 FDESC

A 'first' description of an object. Used in place of LDESC until the object is 'touched' for the first time. 'Touch', in this sense, refers to the state of the TOUCHBIT (described earlier). The PUT and TAKE actions set the TOUCHBIT of an object.

```
(FDESC "First description")
```

12.4 ACTION

Identical to the ACTION property described for ROOMs (remember that OBJECT ACTIONs take no arguments).

```
(ACTION routine-name)
```

12.5 IN

The container of the OBJECT. Described as follows:

```
(IN object-name)
```

12.6 SIZE

The size of the OBJECT, specified this way:

```
(SIZE number)
```

The default size of an object is five on an arbitrary scale (this is changeable although nobody has had any reason to since the scale is arbitrary). A player can carry only so many objects before reaching his load limit, which is the value of the GLOBAL ATOM LOAD-MAX. Routinely, this is one hundred, but 'feel free' to change it.

12.7 CAPACITY

A must for the well-dressed container. Describes the maximum total of the sizes of all objects contained within. The PUT action checks this.

```
(CAPACITY number)
```

Note that the games do not currently distinguish weight and bulk. All are subsumed under SIZE.

12.8 VALUE

For scored games, the VALUE indicates the increment in score for picking up the OBJECT.

```
(VALUE number)
```

12.9 OBJDESC

Indicates that a special ROUTINE is used to describe the object. This is discussed fully later.

```
(OBJDESC routine-name)
```

12.10 CONTFCN

Indicates a special ROUTINE to call for objects contained in this object during the main loop (see the section on the main loop).

```
(CONTFCN routine-name)
```

12.11 New Properties

The properties mentioned above are not the only ones possible. In fact, it was necessary to invent a few for DEADLINE which allowed characters to move around of their own free will. There are limits to the number of different properties allowed in the INTERLOGIC system (namely, thirty-one). No game thus far has used more than twenty-nine. If you want to add a new property, ask either Dave or Marc.

12.12 Manipulating Properties in ROUTINEs

It is possible to get a hold of OBJECT properties from within ROUTINEs as follows:

```
<GETP object property-designator>
```

where 'object' represents any OBJECT and 'property-designator' is the value of a GLOBAL ATOM whose name is the letter 'P' followed by a question mark followed by the name of the property. For example,

```
<GETP ,RUSTY-KNIFE ,P?LDESC>
```

would retrieve the LDESC property of the RUSTY-KNIFE.

To change an OBJECT property, use PUTP as follows:

```
<PUTP object property-designator value>
```

where the first two arguments as in GETP and 'value' being almost anything. For example,

```
<PUTP ,RUSTY-KNIFE ,P?VALUE 0>
```

would make the RUSTY-KNIFE's VALUE property be zero.

Now that this has been said, it should be pointed out that some properties, notably the Exits, cannot be manipulated in this way. However, it is only rarely necessary to do so. Happily, the ACTION, SIZE, CAPACITY, VALUE, and DESC properties behave normally.

12.13 Advanced Property Manipulation

This subsection is not at all necessary for the beginning implementor; feel free to skip it.

The limitation on the properties which can be manipulated with GETP and PUTP derives from the fact that some properties have values which cannot be represented in 16 bits. The information required in a C-exit, including a ROOM, GLOBAL ATOM, and STRING, requires 2-5 bytes. Internally, property values can be from one to eight bytes; only those of one or two bytes can be GETP'ed and PUTP'ed. It is possible, however, to get a TABLE which is the property's value.

```
<GETPT object property-designator>
```

returns a 'property table', which can be manipulated using the subroutines GET and PUT. The size of a 'property table' (i.e. the number of bytes it takes to represent the property) can be checked using PTSIZE:

```
<PTSIZE property-table>
```

Two other subroutines should be described here, since they have no other place in this manual: GETB and PUTB. These are similar to GET and PUT, except that they deal in bytes rather than words (a word is two bytes).

```
<GETB table fix>
```

retrieves the fix'th element of the table (zero based, like GET). PUTB is the inverse.

```
<PUTB table fix value>
```

It turns out that Exits are variable length properties which are checked by the WALK action using GETPT, PTSIZE, and GETB (details will be included at some point.) Similarly, SYNONYM and ADJECTIVE properties are variable length. As an aside, it should be noted that the restriction of SYNONYMs to four and ADJECTIVEs to eight are a direct result of the maximum property length being eight bytes (a SYNONYM requires two bytes to describe, an ADJECTIVE one).

13 Object Descriptions

There are numerous options for describing OBJECTs within a ROOM. Objects are described during the "LOOK" command or when a room is entered (depending on the state of verbosity, whether the room has been previously seen, etc.) A few methods of description have been mentioned already, but this will be a complete listing:

NDESCBIT Setting NDESCBIT for an OBJECT tells the object printer not to print any description of the object. This usually means that the room description already describes the object fully.

LDESC Supplying an LDESC property indicates the long description to be printed for the object. Remember that there is a default LDESC, namely "There is a shmoo here." For size considerations, LDESC's should be used only when necessary. An LDESC of "A shmoo is here." is fairly wasteful, since the default is sufficient.

FDESC Supplying a FDESC property causes a special initial description to be printed for the object. This first description will be printed until the TOUCHBIT of the object becomes set (the verbs take and put do this). This is used for "On hooks above a mantlepiece hangs a sword of great antiquity." and the like, where the initial description of an object is more interesting than simply "There is a sword here."

DESCFCN Supplying a OBJDESC property indicates that a specified ROUTINE is to be used in describing the object. This, naturally, is the most general form of object description, since the ROUTINE can be arbitrarily complex. The ROUTINE so specified is called with one argument whose value is the value of the GLOBAL ATOM M-OBJDESC (another context code!).

14 Movement between ROOMs and Vehicles

In the game environment, movement is usually handled by processing a command like "S", or "WALK NORTH". In these cases, the WALK default handler (assuming no Location ROUTINE handles WALK...) looks for the appropriate property (e.g. NORTH, SOUTH) in the ROOM and, using the method described earlier for Exits, determines whether or not the movement succeeds. If it does, the ACTOR is MOVEd to the new ROOM and the GLOBAL ATOM HERE is SETG'ed to that ROOM. In addition, the new ROOM's ACTION ROUTINE (if any) is called with a context code which is the value of the GLOBAL ATOM M-ENTER. This allows a ROOM to do something special whenever it is entered. The result of the call to the ACTION ROUTINE is ignored (unless it returns 'fatal').

'Vehicles', in the INTERLOGIC sense, are objects (rather than rooms) which can be the location of the ACTOR. In the ZORK games, there was a magic boat, a magic balloon, a time machine, etc. which all fit this definition. In STARCROSS, there is the control couch and bunk. Vehicles must have the VEHBIT set. Each room, as mentioned earlier, needs to have a flag set which describes the 'medium' (for lack of a better word) in which travel is possible. The RLANDBIT describes land-based travel (i.e. walking). Other flags in the ZORK games have been RWATERBIT (for river rooms) and RAIRBIT (for volcano core rooms).

The WALK handler checks the 'medium' in determining whether it is possible to legally go from one room to another. If the current room is on land, for example, and the destination is on water then the movement fails UNLESS one is in a vehicle which is equipped for the 'medium' of the destination. The 'vehicle type' is defined in the OBJECT description by using the VTYPE property (not previously described). The VTYPE property is set up specially and either Dave or Marc should be consulted if it is desired to use vehicles.

Movement is restricted within vehicles by use of the Location ROUTINE (i.e. the ACTION property of the vehicle). Remember that this ROUTINE takes one argument (a context code). Since this is also an OBJECT ROUTINE, the argument should be declared as optional with a default of FALSE.

```
<ROUTINE COUCH-F ("OPTIONAL" (RARG <>)) ...>
```

The ACTION for the control couch, for example, doesn't allow WALKing ("You are seated...") or TAKEing things ("You can't reach the...") It also handles 'getting up' if the seat belt is fastened ("You are belted in...").

One interesting note about ROOMs: they can be destroyed arbitrarily within the context of a game. This is done by setting the RMUNGBIT of the ROOM and changing the LDESC of the ROOM to be something appropriate to print whenever the ACTOR attempts to enter the ROOM. The WALK action checks for the RMUNGBIT and prints the appropriate message.

15 Queued Actions

It is possible to 'queue' any ROUTINE to occur after any future number of 'moves' has passed. To queue a ROUTINE, say:

```
<ENABLE <QUEUE routine-name fix>>
```

where 'routine-name' is the name of the ROUTINE to call and 'fix' is the number of moves after which it will be called. Note that a second argument of one will cause the ROUTINE to be called on the same move. By convention, all queued action ROUTINEs have names beginning with "I-". For example,

```
<ENABLE <QUEUE I-SNAP 10>>
```

To 'turn off' a queued action, use:

```
<DISABLE <INT routine-name>>
```

Yes, the 'user interface' to queued actions is not 'friendly'. We do what we can, but...

It is frequently desired to cause a ROUTINE to be called after every move (at least for a while). This is done by specifying -1 as the second argument to QUEUE. Such ROUTINEs should have some way of turning themselves off when no longer required.

Queued actions can cause some silly side-effects if they are not turned off at appropriate times (especially when the player 'dies'). To handle this case, a ROUTINE called KILL-INTERRUPTS exists which is a series of <DISABLE <INT ...>> expressions. It is called by the ROUTINE JIGS-UP which is used to 'kill' the ACTOR.

One common mistake in writing a queued action is not checking on the PLAYER's location before doing or printing something. For example, one might have a queued action tell the player that some being has left his cage. It would be strange indeed if that message were printed while the player were nowhere near the cage.

16 Actors

Actors are OBJECTs which represent computers, beings, etc. which can be addressed and which can perform actions. They are defined as are other OBJECTs and require an ACTION ROUTINE. Remember that the ACTOR ROUTINE occurs first in processing the player's input and remember also that the OBJECT ROUTINE takes no arguments. This can lead to some confusion, since the ACTION ROUTINE will be called in two cases: first, as either the direct or indirect object and second, as the ACTOR.

These two cases can be distinguished by a check for whether the value of the GLOBAL ATOM WINNER is the actor's OBJECT, e.g.

```
<COND (<EQUAL? ,WINNER ,COMPUTER> ...) ...>
```

If the check succeeds, then this is the case of the ACTOR acting, rather than the player. Since the result of allowing any old command to percolate down to the default action handlers will lead to confusing results (the default handlers all expect that the player is the ACTOR), the ACTOR's ACTION ROUTINE should handle every input, defaulting the uninteresting cases to printing something like "I'm only a stupid robot and can't do that!". Remember that references to the game player should be to the value of the GLOBAL ATOM PLAYER and not WINNER, since WINNER is always the current ACTOR!

17 Syntaxes

The legal input syntaxes are not 'hard-wired', but rather are all specified in a syntax file. These can be manipulated at will and new verbs and types of sentences can be created with great ease, providing they don't do anything fancier than verb/direct object/indirect object.

Syntaxes are defined using the subroutine SYNTAX. The start of a syntax definition is entering a sentence with the token OBJECT in the place of a noun phrase, and putting all prepositions in place:

```
<SYNTAX PUT OBJECT IN OBJECT ....>
```

This indicates that any sentence "PUT noun-phrase IN noun-phrase" is a legal syntax. The end of the definition is an equal-sign followed by the name of the default hander and default pre-action handler (if any). Thus,

```
<SYNTAX PUT OBJECT IN OBJECT = V-PUT PRE-PUT>
```

By convention, the name of the default handler is "V-" followed by the name of the action and the name of the pre-action handler is "PRE-" followed by the name of the action.

Since the parser, in its wisdom, ignores the second preposition of two consecutive ones, the following will work:

```
<SYNTAX SIT DOWN OBJECT = V-SIT>
```

Note that "SIT DOWN CHAIR" and "SIT DOWN ON CHAIR" would both be handled by this case. Here are some more typical looking syntax definitions:

```
<SYNTAX LOOK = V-LOOK>

<SYNTAX TAKE OBJECT = V-TAKE PRE-TAKE>

<SYNTAX TAKE OFF OBJECT = V-REMOVE>

<SYNTAX TAKE OBJECT OUT OBJECT = V-TAKE PRE-TAKE>
```

Notice first that "TAKE OFF X" is a different action than "TAKE X". Also note that the last example handles the case of "TAKE X OUT OF Y".

Although this is all that is required for creating syntaxes, often it is desirable to tell the parser more about the noun-phrases. This can be done by including a LIST with any of the following tokens immediately following the OBJECT token:

TAKE Indicates that the parser should attempt to take the object (the READ action and actions requiring tools often indicate this).

HAVE Requires that the object be in the player's possession. If not, the parser prints "You don't have the x." and no further processing is done. It is exactly as if the parser failed.

MANY Indicates that multiple objects can be used with this syntax. The various TAKE and PUT syntaxes have this token.

Other tokens are used to indicate where the parser is to look for objects specified in the noun-phrases. The default (which was mentioned earlier) is to look in the ROOM and ACTOR. Often, however, this default is not very clever. For example, the TAKE syntaxes probably aren't interested in objects that are already held. Here are the appropriate tokens:

HELD Examine things held 'at top level' (i.e. not contained within something else being held.)

CARRIED Examine things held at other than top level (i.e. things contained within things held.)

ON-GROUND Examine things on the ground 'at top level' (i.e. not contained in things on the ground.)

IN-ROOM Examine things contained within things on the ground.

As has just been noted, the default is ALL of the above. One TAKE syntax is specified this way:

```
<SYNTAX TAKE OBJECT (MANY ON-GROUND)
= V-TAKE PRE-TAKE>
```

This indicates that more than one object can be specified in the sentence and that objects should only be examined which are on the ground at top level. The default handler and pre-action handler are specified.

The final piece of confusion involves what is known as GWIM'ming (GWIM stands for 'get what I mean'), the procedure by which the game seems to magically assume a certain unspecified object. This works by including a LIST whose first element is the token FIND and the second is the name of a condition flag after the OBJECT token. If no object is specified in that position, the parser will look for one with that condition flag. For example,

```
<SYNTAX OPEN OBJECT (FIND DOORBIT) = V-OPEN>
```

If the player says simply "OPEN", this will work providing there is only one object accessible with the DOORBIT set.

18 Verb Synonyms

It is very straightforward do create verb synonyms using the subroutine SYNONYM. These are usually placed after the syntaxes for a verb. The format is:

```
<SYNONYM TAKE CARRY GET HOLD>
```

This indicates that the words CARRY, GET, and HOLD are all synonyms for the word TAKE.

Verb synonyms are dangerous in two respects:

- The subroutine VERB? refers to the 'master' verb only. It is an error to say <VERB? CARRY> since CARRY is only a synonym of the 'real' verb, TAKE. This confusion is regrettable and will be fixed in the future. Most of the 'real' verbs are obvious. Others, like MUNG being the 'real' verb for DESTROY, BREAK, etc. are not obvious and lead to problems.
- They must be used with great caution if they are also another part of speech. This problem occurred in STARCROSS with the word HOOK, which was made a synonym of TIE. Unfortunately, HOOK is also a noun in STARCROSS. The results were peculiar. Speak to Dave or Marc if a problem like this arises.

19 Game Files

It is possible to think of a game as containing a 'substrate' of unchangable parts and a 'script' of changable ones. However, since even such 'substrate' items as the main loop and parser are subject to change from one game to another the term is a bit fuzzy. Here are the names of files which exist for each INTERLOGIC game. Note that the second file name (extension) of all game files is ZIL.

game Where 'game' is the name of the game. This is a file which loads the remining files.

MAIN Contains the main loop and related utilities. The ROUTINE GO (at which the game starts) is located here.

CLOCK Contains the queued action driver

PARSER Contains the parser code

SYNTAX Contains syntaxes defined for the game

MACROS Contains macros (this doesn't change from game to game)

VERBS Contains default handlers and pre-action handlers

DUNGEON Contains room and object definitions

ACTIONS Contains other game-dependent ROUTINEs

The last two files are a subject of some debate. It has been the common practice to split the script into these two files; however, this is a historical result of the old development system and need no longer be followed. ZORK III tries a new approach, splitting the game into areas, each of which corresponds to a file. In that file are OBJECTs, ROOMs, and ROUTINEs related to that area of the game. The file DUNGEON is still used for GLOBAL OBJECTs and some other definitions. There is no hard and fast rule on this. However, I would recommend the newer system; it allows you to define OBJECTs and write their ACTION ROUTINEs in the same place. (Since this was written, all games have used the "new" method.)

20 Useful Utility ROUTINEs

The following are a bunch of utility ROUTINEs which are quite useful:

20.1 PERFORM

PERFORM takes three arguments, corresponding to the values of the GLOBAL ATOMs PRSA, PRSO, and PRSI (the last two are optional). It calls the internal handler sequence which was discussed early on (see Main Loop) AS IF its arguments were the parser's output. In other words, it simulates another input. This can be used to avoid having identical code in a number of different places. Since the result of calling the handler sequence always involves doing SOMETHING (see Philosophy of Main Loop), the call to PERFORM should always be followed by an <RTRUE>. The form of a PERFORM call is:

```
<PERFORM ,V?verb object-1 object-2>
```

For example,

```
<PERFORM ,V?TAKE ,RUSTY-KNIFE>
```

This would simulate an input of "TAKE RUSTY KNIFE".

20.2 ROB

ROB takes two arguments, both OBJECTs, and moves all of the contents from one to the other.

```
<ROB ,WINNER ,WIZARDS-CASE>
```

would move all of the WINNER's possessions to the WIZARDS-CASE.

20.3 RANDOM

RANDOM, given a FIX, returns an integer between one and that FIX, inclusive, randomly.

20.4 PROB

PROB, given a FIX from one to one hundred, returns 'true' FIX percent of the time, randomly. PROB is approximately <NOT <L? .FIX <RANDOM 100>>>.

21 Commonly Used GLOBAL ATOMs

Here are some commonly used GLOBAL ATOMs, and their significance:

WINNER The current ACTOR (i.e. the OBJECT which is performing tasks in the game sense).

PLAYER The OBJECT representing the flesh-and-blood player. The values of WINNER and PLAYER are the same, except in the cases where another ACTOR is performing a task.

SCORE The current score.

SCORE-MAX The maximum score (used by the SCORE action).

MOVES The number of moves the player has taken during this session. It is incremented in the main loop.

HERE The ROOM which WINNER occupies.

LIT Whether or not the current environment is lighted.

PRSO, PRSI Direct and Indirect object (parser output).

PRSA Action (parser output). This GLOBAL is 'hidden' in that the VERB? subroutine handles the common use for examining its value.

P-CONT Can be SETG'ed to 'false' to indicate that no further commands on an input line should be processed.

22 The New "Takenology" – SEM 10/19/83

In the first version of the new takenology, a DONTTAKEBIT was needed. This is no longer the case. The TAKEBIT and the TRYTAKEBIT, which already exist, do everything that needs to be done.

Things with only the TAKEBIT will get implicitly taken, and TAKE ALL will attempt to take them. In other words, READ OBJECT does a "(Taken)", and TAKE ALL will say "Object: Taken." Most takeable objects that are just lying innocently around should fall into this category.

Any object with the TRYTAKEBIT will not be implicitly taken, but TAKE ALL will try to take it. If object has both the TAKEBIT and the TRYTAKEBIT, it means that you want to make sure that the object doesn't get taken without clearing it through the objects action routine. This is meant for things that you don't want READING, EATING, etc. to take automatically, such as the royal jewels in Zork III, or the key in the crevice in Planetfall. But you still want TAKE ALL to try for that object.

Things with just the TRYTAKEBIT are objects which are never takeable, but which some players might conceivably think are takeable, so that TAKE ALL will try to take them. They should have something in their action routine that handles TAKE, like "Ooomph, its alot heavier than it looks" or "It's too big to carry but you might try pushing it". Examples of objects with just the TRYTAKEBIT are the carpet and the mailbox in Zork I.

Things with neither the TAKEBIT or the TRYTAKEBIT are neither implicitly taken or noticed by TAKE ALL.

Recapping the rules:

- All takeable objects should have the TAKEBIT.
- Any object whose action routine handles TAKE should have the TRYTAKEBIT.
- If the TRYTAKEBIT is used to handle some initial condition (such as the red rod in the rat-ant nest in Starcross) it should be FCLEARed when that condition no longer applies (in this case, when it is removed from the nest). Similarly, one can easily imagine conditions where the TRYTAKEBIT would be FSET during the game.
- TAKE ALL will try to take any object with either the TAKEBIT or the TRYTAKEBIT.
- If it isn't already obvious, the TRYTAKEBIT has two different meanings depending on whether the object also has the TAKEBIT. If the object doesn't have the TAKEBIT, then it means "take all me". If it does have the TAKEBIT, then it will be take-alled anyway, and the TRYTAKEBIT is only to tell the parser not to implicitly take me.

There is another aspect of the new takenology, which involves containers which should behave like surfaces with respect to TAKE ALL. Normally, TAKE ALL will not attempt to take anything in a container. However, some containers are more like surfaces, and should be take-alled-from. An example of this is the fire pit in Infidel. (Note: the reason that the fire pit is a container rather than a surface is that we want the describers to say that things are IN the fire pit, rather that ON it.) For details about this aspect of takenology, consult MSB.

Here is a table to cover the four combinations of these two flags:

WHICH BITS?	NORMAL TAKE RESP.	TAKE ALL?	IMPLICIT TAKE?
Neither	"What a concept."	ignore it	no
Just TAKEBIT	"Taken."	attempt it	yes
Just TRYTAKE	"You can't ever take this object."	attempt it	no
Both	"You can't take this object now."	attempt it	no